Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Kolar

Implementacija in razširitve metode SCITE za bayesovsko modeliranje mutacijskih dreves

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Erik Štrumbelj
Somentor: prof. dr. Blaž Zupan

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Študent naj prouči in v jeziku Python implementira metodo SCITE za bayesovsko modeliranje mutacijskih dreves. Nato naj izboljša računsko učinkovitost algoritma z optimizacijo delov programske kode in/ali bolj učinkovitim sprehajanjem po prostoru aposteriorne porazdelitve modela z MCMC. Za potrebe slednjega naj implementira tudi neko smiselno oceno, kako učinkovit je sprehod MCMC po prostoru dreves.

Zahvaljujem se svojemu mentorju izr. prof. dr. Eriku Štrumblju, ki mi je med izdelavo diplomske naloge pomagal z razlago konceptov Bayesove statistike in s konstruktivnimi komentarji glede možnih optimizacij metode SCITE. Hvaležen sem tudi somentorju prof. dr. Blažu Zupanu, ki je prispeval k mojemu razumevanju biološkega ozadja diplomske naloge. Nazadnje se zahvaljujem as. Martinu Stražarju za pomoč pri pridobitvi novih podatkov.

Idi.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Metoda SCITE **3**

2.1 Problematika vzorčenja 7

3 Implementacija v okolju Python **9**

3.1 Podrobnosti implementacije 10

3.2 Delno ocenjevanje mutacijskih dreves 11

3.3 Razširitev programske opreme z izračunom ESS 16

3.4 Nov premik drevesa 18

4 Poskusi in rezultati **23**

4.1 Podatki 23

4.2 Pravilnost implementacije 24

4.3 Delno ocenjevanje mutacijskih dreves 27

4.4 Hitrost implementacije 28

4.5 Uporaba novega premika 28

4.6 Učinkovitejša implementacija SuMoTED 31

5 Zaključek **33**

Seznam uporabljenih kratic

kratica	angleško	slovensko
MAP	maximum a posteriori	maksimum aposteriorne porazdelitve
ML	maximum likelihood	največje verjetje
BFT	breadth first traversal	obhod drevesa pri iskanju v širino
DFT	depth first traversal	obhod drevesa pri iskanju v globino
MCMC	Markov chain Monte Carlo	markovske verige Monte Carlo
ESS	effective sample size	efektivno število vzorcev
SCS	single-cell sequencing	sekvenciranje posameznih celic

Povzetek

Naslov: Implementacija in razširitve metode SCITE za bayesovsko modeliranje mutacijskih dreves

Avtor: Luka Kolar

Metoda SCITE lahko rekonstruira potek razvoja rakavih obolenj v celicah iz podatkov o njihovih mutacijah. Potek razvoja predstavi z mutacijskim drevesom, ki je sorodno filogenetskemu drevesu. V diplomski nalogi implementiramo del funkcionalnosti metode v programskem jeziku Python in zagotovimo primerljivo hitrost delovanja. Osredotočimo se na pridobitev aposteriornih porazdelitev mutacijskih dreves ter verjetnosti nezaznanih mutacij v celicah. Metodo izboljšamo z delnim ocenjevanjem mutacijskih dreves, ki omogoča hitrejše ocenjevanje. Poleg obstoječih premikov mutacijskih dreves predlagamo nov premik in na več podatkovnih množicah utemeljimo njegovo uporabnost. Nazadnje uporabniku omogočimo izračun efektivnega števila vzorcev, s katerim lahko bolje ovrednoti rezultate izvajanja algoritma.

Ključne besede: mutacijska drevesa, rakava obolenja, algoritem Metropolis–Hastings, metoda SCITE.

Abstract

Title: Implementation and extensions of the SCITE method for Bayesian modelling of mutation trees

Author: Luka Kolar

SCITE method can reconstruct the course of development of cancer in cells from data on their mutations. The course of development is represented with a mutation tree, which is similar to a phylogenetic tree. In the thesis, we implement some functionalities of the SCITE method with the Python programming language and ensure comparable execution time. We focus on the posterior distributions of mutation trees and the probabilities of overlooked mutations in cells. The method is improved with the introduction of partial mutation tree scoring which speeds up the scoring process. Along the existing tree moves we propose a new tree move and prove its usefulness on multiple datasets. Lastly, we enable the user to compute the effective sample size of posterior samples and thus enable better assessment of the results of the algorithm.

Keywords: mutation trees, cancer, Metropolis–Hastings algorithm, SCITE method.

Poglavje 1

Uvod

O evolucijski zgodovini tumorja se lahko sklepa na podlagi celic vzorčenih iz tumorja. Vzorec predstavlja trenutni posnetek stanja v tumorju, iz katerega ni vedno moč neposredno ugotoviti zgodovine [4]. Dobro razumevanje tumorja, h kateremu pripomore tudi poznavanje evolucijske zgodovine, je pomembno pri pristopih k zdravljenju rakavih obolenj, ki se osredotočajo na vsakega pacienta posebej [11, 12].

V diplomski nalogi se osredotočimo na metodo SCITE [4], ki jo podrobneje opišemo v poglavju 2. Med pisanjem diplomske naloge so avtorji metode implementirali nove funkcionalnosti in razširjeno metodo poimenovali ∞SCITE[1] [6]. Nova metoda poleg verjetnosti nezaznanih mutacij (angl. *false negative*) lahko ugotavlja tudi verjetnost napačno zaznanih mutacij (angl. *false positive*). Razširitev za poljubno mutacijo omogoči, da se pojavi dvakrat oziroma da mutacija izgine. S tem se delno izogne predpostavki o enkratni pojavitvi vsake mutacije iz originalne metode. Modelirajo pa se lahko tudi primeri, v katerih je več celic pri sekvenciranju zaznanih kot ena sama celica (angl. *doublet samples*). Metoda, ki sta jo razvila Kim in Simon [5], temelji na razvrščanju parov mutacij in tako kot SCITE vrača mutacijska drevesa. BitPhylogeny [15] k problemu pristopa povsem bayesovsko. Metoda ugotavlja, kako združiti celice v gruče ter katero razvojno drevo najbolj

[1] https://github.com/cbg-ethz/infSCITE

verjetno povezuje gruče celic. OncoNEM [9] za razliko od SCITE namesto mutacijskih dreves uporablja drevesa, pri katerih vozlišča predstavljajo celice. Najboljša drevesa išče z obiskovanjem dreves v okolici do sedaj najboljših obiskanih dreves. Poleg vzorčenih celic drevesom dodaja tudi celice, ki niso bile vzorčene, po končanem procesu iskanja najboljših dreves pa celice v drevesu združi v gruče. SiFit [16] podobno kot metoda ∞SCITE odpravlja predpostavko o enkratni pojavitvi vsake mutacije, ki je prisotna pri metodi SCITE. ddClone [10] o dogajanju v tumorju sklepa tako z uporabo podatkov, pridobljenih z metodo sekvenciranja posameznih celic (angl. *single-cell sequencing, SCS*), kot tudi z uporabo podatkov, pridobljenih z množičnim sekvenciranjem (angl. *next-generation sequencing, NGS*). Vse omenjene metode sicer uporabljajo podatke, pridobljene z metodo SCS.

Cilj diplomske naloge je razumeti in v programskem jeziku Python implementirati metodo SCITE [4]. Hitrost delovanja algoritma je ključnega pomena, zato si v diplomski nalogi prizadevamo implementirati algoritem čim bolj učinkovito, kar pa zaradi uporabe programskega jezika Python namesto programskega jezika C++, ki so ga uporabili avtorji metode, predstavlja izziv. Naš cilj je tudi razširiti metodo z dodatnimi funkcionalnostmi, ki pripomorejo k učinkovitejšemu ugotavljanju dogajanja v tumorju.

V besedilu, ki sledi, najprej predstavimo metodo SCITE, nato opišemo implementacijo v programskem jeziku Python in razširitve metode, zaključimo pa z ovrednotenjem naše implementacije ter razširitev.

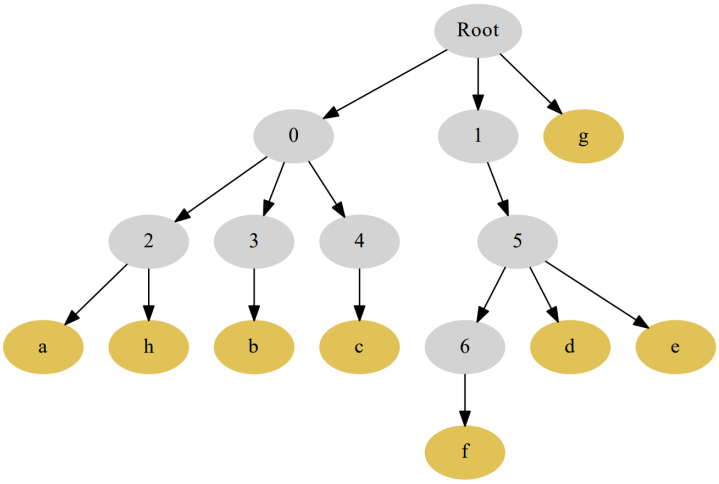
Poglavje 2

Metoda SCITE

Sledeče poglavje povzema članek o metodi SCITE [4] in njej pripadajočo programsko opremo[1]. SCITE (**S**ingle **C**ell **I**nference of **T**umor **E**volution) omogoča vpogled v evolucijsko zgodovino tumorja iz šumnih in nepopolnih mutacijskih profilov posamičnih celic. Vzorci celic, ki jih metoda analizira, so pridobljeni z metodo SCS.

Rezultat izvajanja algoritma je mutacijsko drevo (angl. *mutation tree*), ki pojasnjuje vzorčene mutacije v celicah. Mutacijsko drevo je drevo, pri katerem vozlišča drevesa predstavljajo mutacije. Odnos med starši in otroci drevesa modelira časovno zaporedje mutacij, pri čemer se je mutacija v starševskem vozlišču zgodila pred tisto v vozlišču otroka. Prostor možnih mutacijskih dreves avtorji omejijo s predpostavko, da se je vsaka zaznana mutacija zgodila natanko enkrat (angl. *infinite sites assumption*). S to predpostavko se posledično predpostavlja tudi, da so vse mutacije heterozigotne. Homozigotne mutacije so sicer lahko zaznane, a so modelirane kot posledica napake pri vzorčenju celic, saj bi sicer nakazovale na dvakratno pojavitev iste mutacije, kar pa nasprotuje osnovni predpostavki. Mutacijska drevesa so sorodna filogenetskim, saj množica celic, ki je pripeta na isto vozlišče v mutacijskem drevesu, predstavlja takson v filogenetskem drevesu. Na sliki 2.1 je primer mutacijskega drevesa s pripetimi celicami.

[1] https://github.com/cbg-ethz/SCITE



Slika 2.1: Primer mutacijskega drevesa (siva vozlišča) s pripetimi celicami (rumena vozlišča). Korensko vozlišče `Root` za razliko od ostalih vozlišč ne predstavlja nobene mutacije; pripetje celice nanj pomeni, da najverjetneje na to celico ni vplivala nobena mutacija.

Metoda SCITE lahko iz podatkov ugotavlja tudi verjetnost, da določena mutacija v celici ni zaznana, čeprav je v njej prisotna. V metodi je ta verjetnost označena z β. Verjetnost obratnega pojava, pri katerem mutacija v celici ni prisotna, a je vseeno zaznana, je označena z α. Slednje verjetnosti metoda iz podatkov ne zna ugotavljati in jo mora zato specificirati uporabnik. Če uporabnik pozna tudi verjetnost β, je algoritem ne ugotavlja.

Analitična rešitev problema za šumne in nepopolne podatke ne obstaja, zato avtorji uporabijo algoritem Metropolis–Hastings, ki je eden izmed metod markovskih verig Monte Carlo (angl. *Markov chain Monte Carlo, MCMC*). Algoritem obiskuje stanja markovske verige, pri kateri vsako stanje predstavlja kombinacijo mutacijskega drevesa in β oziroma zgolj mutacijsko drevo ob poznavanju β. Pri metodi SCITE algoritem obiskuje sosednja stanja, ki se od trenutnega razlikujejo samo v mutacijskem drevesu ali β, ne pa v obeh hkrati. Verjetnost izbire spremembe β namesto mutacijskega drevesa je eden izmed parametrov metode in je enak 0, če je β znana.

V primeru spremembe mutacijskega drevesa se najprej določi tip premika mutacijskega drevesa, nato pa še parametri izbranega premika. Metoda ponuja tri premike mutacijskih dreves:

1. „Prune and reattach“ iz mutacijskega drevesa odstrani naključno poddrevo in ga pripne na poljubno vozlišče v drevesu.

2. „Swap node labels“ dvema naključnima različnima vozliščema (korensko vozlišče ne more biti izbrano) zamenja oznaki. To je edini izmed premikov, ki vedno ohrani strukturo mutacijskega drevesa.

3. „Swap subtrees“ dvema naključnima poddrevesoma zamenja starša. V primeru, da je prvo poddrevo naslednik drugega, se prvo poddrevo pripne na starša drugega poddrevesa, drugo poddrevo pa na poljubno vozlišče prvega. V tem primeru je potreben tudi izračun korekcijskega koeficienta, ki ga opišemo kasneje.

Pri spremembi β pa se trenutni β prišteje slučajna spremenljivka

$$X \sim \mathcal{N}\left(0, \left(\frac{\sigma_\beta}{x_\beta}\right)^2\right), \tag{2.1}$$

pri čemer sta σ_β in x_β parametra metode. Parameter σ_β skupaj s parametrom μ_β opisuje apriorno porazdelitev β. Parametra podrobneje opišemo kasneje.

V začetnem stanju je mutacijsko drevo naključno, β pa je nastavljena na μ_β. Naključno mutacijsko drevo se dobi z generiranjem naključne Prüferjeve kode, ki se nato pretvori v drevo.

Označimo trenutno stanje s S, generirano sosednje stanje s S_{new}, verjetnost izbire sosednjega stanja glede na trenutno stanje z q, funkcijo ocene stanja pa s F. Algoritem se v S_{new} premakne z verjetnostjo

$$min\left\{1, \frac{q(S|S_{new})F(S_{new})^\gamma}{q(S_{new}|S)F(S)^\gamma}\right\}, \tag{2.2}$$

sicer ostane v istem stanju. Parameter γ vpliva na razgibanost porazdelitve in lahko pripomore k hitrejšemu odkritju boljših stanj. Nastavi ga lahko

uporabnik, sicer pa ima privzeto vrednost 1. V primeru spremembe β in uporabe premikov mutacijskih dreves „prune and reattach“ ter „swap node labels“ vedno velja $q(S|S_{new}) = q(S_{new}|S)$, zato poznavanje q v teh primerih ni potrebna. Pri „swap subtrees“ pa lahko v primeru, da je prvo poddrevo naslednik drugega velja $q(S|S_{new}) \neq q(S_{new}|S)$. V tem primeru je potreben izračun korekcijskega koeficienta:

$$c = \frac{q(S|S_{new})}{q(S_{new}|S)}. \tag{2.3}$$

Metoda omogoča iskanje dveh tipov dreves. Pri iskanju dreves ML se pri ocenjevanju drevesa izbere le najboljše pripetje celic na vozlišča mutacijskega drevesa. Ker ocena ne predstavlja aposteriorne verjetnosti stanja, posledično tudi obiskana stanja markovske verige ne predstavljajo aposteriorne porazdelitve mutacijskih dreves in β. V tem primeru algoritem Metropolis–Hastings zgolj narekuje način pregledovanja prostora možnih kombinacij mutacijskih dreves in β. Ocena $F(S)$ je sestavljena iz ocene oziroma verjetnosti β in ocene mutacijskega drevesa (T). Iz obeh ocen se enostavno izračuna kot

$$F(S) = P(\beta)F(T). \tag{2.4}$$

Omeniti velja, da je ocena $F(T)$ odvisna od α in β.

Verjetnost $P(\beta)$ se izračuna kot verjetnost trenutne β glede na apriorno porazdelitev β. Apriorno porazdelitev opisuje porazdelitev beta, njene parametre pa se izračuna iz aritmetične sredine in standardnega odklona (μ_β in σ_β), ki ju specificira uporabnik.

Metoda pri iskanju dreves ML ponuja tudi uporabo binarnih genealoških dreves (angl. *binary genealogical tree*). Pri omenjenih drevesih notranja vozlišča nimajo pomena, listi dreves pa predstavljajo vzorčene celice. Mutacije so zavedene v povezavah drevesa namesto v vozliščih.

Pri iskanju dreves MAP ocena $F(S)$ predstavlja aposteriorno verjetnost stanja:

$$F(S) = P(S|D) = \frac{P(D|S)P(S)}{P(D)} \propto P(D|S)P(S). \tag{2.5}$$

Za pravilno delovanje mora biti parameter γ v enačbi (2.2) nastavljen na 1. Z D tukaj pojmujemo vhodno matriko dimenzij $n \times m$, v kateri so stolpci mutacijski profili. Parameter m torej označuje število vseh celic, parameter n pa število različnih mutacij. Komponenta D_{ij} označuje, kako je bila zaznana i-ta mutacija v j-ti celici, pri čemer 0 pomeni odsostnost mutacije, 1 heterozigotno mutacijo, 2 homozigotno mutacijo in 3 manjkajoč podatek. Avtorji metode oceno stanja izpeljejo kot

$$F(S) \propto \prod_{j=1}^{m} \sum_{\sigma_j=1}^{n+1} \left[\prod_{i=1}^{n} P\left(D_{ij} | A(T)_{i\sigma_j}\right) \right] P(\sigma_j|S)P(S). \tag{2.6}$$

Avtorji predpostavljajo enakost $P(S) = P(T)P(\beta)$ ter uniformno porazdelitev $P(\sigma_j|S)$ in $P(T)$, zato se enačba (2.6) poenostavi v

$$F(S) \propto \prod_{j=1}^{m} \sum_{\sigma_j=1}^{n+1} \left[\prod_{i=1}^{n} P\left(D_{ij} | A(T)_{i\sigma_j}\right) \right] P(\beta). \tag{2.7}$$

Verjetnosti tipa $P\left(D_{ij} = x | A(T)_{i\sigma_j} = y\right)$ se da izračunati iz verjetnosti α in verjetnosti β stanja, ki se ocenjuje. Verjetnost $P(\beta)$ pa se izračuna na enak način kot pri iskanju dreves ML. Enačbo za izračun ocene (2.7) nadalje opišemo v poglavju 3.2. Obiskana stanja tvorijo aposteriorni porazdelitvi mutacijskih dreves in β, ki ju metoda vrača ob koncu izvajanja. Primere le-teh lahko vidimo na sliki 4.1. Drevesa in β MAP so tista stanja, ki imajo najvišjo oceno izmed vseh stanj v aposteriorni porazdelitvi. Prvih 25 % vzorcev aposteriorne porazdelitve pripada fazi „burn-in“, ki se jo iz porazdelitve odstrani.

Velja še omeniti, da se pri obeh pristopih končne točke pripetja celic določijo na enak način, kot poteka določitev najboljšega pripetja pri ocenjevanju po pristopu ML.

2.1 Problematika vzorčenja

Stanja, ki jih obišče metoda SCITE [4], so med seboj odvisna, saj je naslednje stanje vedno ali enako prejšnjemu v primeru zavrnitve novega stanja ali

pa je bilo generirano na podlagi prejšnjega. Posledično vzorci, pridobljeni z obiskovanjem stanj, o aposteriorni porazdelitvi povedo manj, kot bi povedali neodvisni vzorci. Pojav se lahko opiše z avtokorelacijo, ki pove stopnjo koreliranosti zaporedja z istim zaporedjem v preteklosti [2]. Pri SCITE govorimo o pozitivni avtokorelaciji, saj so si sosednja stanja podobna; pri negativni avtokorelaciji bi si bila sosednja stanja med seboj različna, a še vedno odvisna med seboj. Visoka stopnja avtokoreliranosti povzroči, da iz vzorcev o želeni porazdelitvi izvemo zelo malo. Pri interpretaciji rezultatov je zato potrebna previdnost, saj veliko število vzorcev ne pomeni nujno tudi, da vemo veliko o aposteriorni porazdelitvi.

Zaradi odvisnosti stanj je tudi potrebna faza „burn-in“. S fazo se začetno stanje verige MCMC nastavi na neko poljubno stanje po določenem številu obiskanih stanj. S tem se odstrani vpliv izbire začetnega stanja [1].

Poglavje 3

Implementacija v okolju Python

Implementirali smo del funkcionalnosti, ki jih ponuja metoda SCITE [4] oziroma njej pripadajoča programska oprema. Pri implementaciji smo se osredotočili na iskanje mutacijskih dreves MAP in β MAP ter pridobitev aposteriornih porazdelitev. Kot pokažemo v poglavju 4.2, naša implementacija v programskem jeziku Python deluje skladno z implementacijo avtorjev metode v programskem jeziku C++. V mnogih pogledih sta si implementaciji podobni, saj specifike metode ne dopuščajo veliko svobode pri implementaciji, prav tako pa je metoda v programskem jeziku C++ implementirana učinkovito, zato v večini primerov ni potrebe po iskanju drugačnih pristopov. Kljub temu smo nekatere funkcionalnosti za potrebe večje učinkovitosti implementirali drugače, poleg tega pa smo programsko opremo avtorjev tudi razširili z dodatnimi funkcionalnostmi.

Programska oprema je na voljo na spletnem naslovu `https://github.com/lukakolar/Python-SCITE`.

3.1 Podrobnosti implementacije

Metoda SCITE [4] pri delovanju uporablja binarno matriko prednikov A (angl. *ancestor matrix*), pri kateri je komponenta A_{ij} enaka 1, če je vozlišče i prednik vozlišča j. Pri matriki predpostavljamo, da je vsako vozlišče prednik samega sebe. Če z n označimo število mutacij, ima matrika n^2 elementov, zato je za njeno izgradnjo potrebnega $O(n^2)$ časa. Mutacijska drevesa imajo sicer $n + 1$ vozlišč, a korensko vozlišče v matriki prednikov programsko ni vključeno, ker vedno velja, da je korensko vozlišče prednik vseh ostalih vozlišč ter da nima prednikov. Kadar govorimo o matriki prednikov s stališča izpeljave, tako dejansko govorimo o matriki dimenzij $(n + 1) \times (n + 1)$. Ob spremembi drevesa se spremenijo le nekateri vnosi matrike A, zato lahko ob že zgrajeni matriki A, novo matriko A dobimo v manj kot $O(n^2)$ časa. V programski kodi metode SCITE v programskem jeziku C++ je omenjena izboljšava implementirana za premika „prune and reattach“ in „swap node labels“, a zaradi nam neznanih razlogov ni uporabljena. Izboljšavi sta uporabljeni v naši implementaciji, pri čemer smo se pri izboljšavi za premik „swap node labels“ zgledovali po tisti iz implementacije v programskem jeziku C++. O pravilnosti delovanja obeh izboljšav smo se prepričali s testi enote.

Poleg omenjenih testov smo pripravili tudi teste enote za ostale funkcionalnosti. Na ta način smo lahko prepričani, da naša implementacija deluje skladno z implementacijo avtorjev metode.

Naša implementacija se za potrebe učinkovitega delovanja opira na programska paketa NumPy[1] in Numba[2]. Prvi nam omogoča učinkovito uporabo vektorjev in matrik, slednji pa prevajanje „just-in-time (JIT)“. Funkcije, ki se med izvajanjem prevedejo, lahko prepoznamo po dekoratorju `@njit`. Dekoratorju nastavimo neobvezen parameter `cache=True`, ki omogoči, da so prevodi funkcij shranjeni lokalno. Posledično se funkcije ne prevajajo ponovno ob vsakem zagonu programa, kar pospeši delovanje programa. Numba lahko kodo prevaja v objektnem načinu in načinu „nopython“. Slednji način

[1] https://www.numpy.org/
[2] https://numba.pydata.org/

omogoči veliko hitrejše delovanje programa, a je za delovanje potrebno spoštovati stroge omejitve glede uporabe funkcionalnosti programskega jezika Python in programskega paketa NumPy. Numba sicer z novimi izidi verzij podpira vedno več funkcionalnosti; z izidom različice 0.39.0 in s tem podporo gnezdenim seznamom se je tako na primer zmanjšala zapletenost naše implementacije.

3.2 Delno ocenjevanje mutacijskih dreves

Del podpoglavja, ki opisuje ocenjevanje mutacijskih dreves po principu MAP, je povzet po članku metode SCITE [4] in implementaciji metode. V nadaljevanju bomo pojma mutacija in vozlišče uporabljali kot enakovredna, saj vozlišče v mutacijskem drevesu predstavlja mutacijo, razen v primeru korenskega vozlišča.

Iz formule (2.7) lahko vidimo, da se mutacijska drevesa po principu MAP ocenjujejo kot

$$\prod_{j=1}^{m} \sum_{\sigma_j=1}^{n+1} \left[\prod_{i=1}^{n} P\left(D_{ij} | A(T)_{i\sigma_j}\right) \right]. \tag{3.1}$$

Kot smo že omenili, so verjetnosti oblike $P\left(D_{ij} = x | A(T)_{i\sigma_j} = y\right)$ odvisne tudi od β stanja, ki se ocenjuje, zato se po formuli (3.1) dejansko hkrati ocenjuje tako mutacijsko drevo, kot tudi β. Za lažje razumevanje formule na kratko opišimo elemente, ki jih še ne poznamo:

- i označuje trenutno mutacijo; j trenutno celico.
- $n + 1$ označuje korensko vozlišče, ki ne predstavlja nobene mutacije.
- σ_j označuje vozlišče, na katero je pripeta celica j.
- $A(T)_{i\sigma_j}$ je element prej omenjene matrike prednikov, ki nam pove, ali je vozlišče i prednik vozlišča, na katero je pripeta celica j, kar pa nam dejansko pove, ali bi glede na trenutno mutacijsko drevo in pripetje celice j na vozlišče σ_j morala celica j vsebovati mutacijo i.

Produkt

$$\prod_{i=1}^{n} P(D_{ij}|A(T)_{i\sigma_j}) \tag{3.2}$$

tako za neko mutacijsko drevo, celico j in njeno pripetje na vozlišče σ_j opisuje verjetnost opaženih mutacij v tej celici. Za korensko vozlišče se izračun (3.2) poenostavi v

$$\prod_{i=1}^{n} P(D_{ij}|A(T)_{i\sigma_j} = 0), \tag{3.3}$$

saj v korenskem vozlišču ni prisotna nobena mutacija. Za vsa ostala vozlišča k izračun produkta oblike (3.2), ki opisuje pripetje celice j na vozlišče k ($\sigma_j = k$) ni potreben, saj se le-ta lahko izračuna iz produkta, ki opisuje pripetje celice j na starša celice k, ki ga označimo s k'. Ker je v vozlišču k prisotna mutacija, ki ga to vozlišče opisuje, v vozlišču k' pa ne, se do želenega rezultata pride z deljenjem produkta s $P(D_{kj}|A(T)_{k\sigma_j} = 0)$ in množenjem s $P(D_{kj}|A(T)_{k\sigma_j} = 1)$. V algoritmu se dejansko operira z logaritemskimi verjetnostmi, zato se operacije množenja in deljenja spremenijo v operacije seštevanja in odštevanja.

Pri takem načinu programskega ocenjevanja mutacijskih dreves lahko pride do numeričnih napak zaradi mnogih prištevanj in odštevanj števil tipa `double` oziroma `float`. Programska oprema v jeziku C++ implementira tudi natančnejše ocenjevanje, pri katerem se ne prištevajo in odštevajo verjetnosti, pač pa števci različnih verjetnosti tipa $P(D_{ij} = x|A(T)_{i\sigma_j} = y)$. Ker velja $x \in \{0, 1, 2, 3\}$ in $y \in \{0, 1\}$, je potrebno vzdrževati 8 števcev. Implementacija v programskem jeziku C++ za obiskovanje vozlišč uporablja BFT, ki zagotavlja, da je starš vozlišča vedno obiskan pred vozliščem.

V naši implementaciji SCITE si pri ocenjevanju pomagamo z matriko, ki jo v programski kodi poimenujemo `fast_matrix`. Omenjena matrika je dimenzij $(n + 1) \times m$ in vsebuje produkte tipa (3.2) za vsako kombinacijo vozlišča v mutacijskem drevesu in vzorčene celice. Matrika se programsko zgradi v treh korakih:

1. Po formuli (3.3) se določijo verjetnosti pripetja vseh celic na korensko vozlišče.

2. Za vsako pripetje celice na vozlišče k, ki ni korensko, se določi sprememba verjetnosti iz $P(D_{kj}|A(T)_{k\sigma_j} = 0)$ v $P(D_{kj}|A(T)_{k\sigma_j} = 1)$.

3. Za vsako pripetje celice na vozlišče k se iz pripetja iste celice na starševsko vozlišče k' in spremembe verjetnosti iz prejšnje točke določi produkt oblike (3.2).

Iz matrike se nato ocena mutacijskega drevesa z uporabo marginalizacije točk pripetja celic na vozlišča izračuna kot

```
np.sum(np.log(np.sum(np.exp(fast_matrix), axis=0))),
```

kar ponazarja zunanji množenje in seštevanje iz formule (3.1).

Skladno z implementacijo v programskem jeziku C++ smo implementirali tudi natančnejše ocenjevanje z uporabo števcev. Pri natančnem ocenjevanju uporabljamo matriko, ki jo v programski kodi poimenujemo `mask_matrix`. Matrika je dimenzij $(n+1) \times m \times 8$, pri čemer sta prvi dve dimenziji skladni z dimenzijami matrike `fast_matrix`, tretja dimenzija pa predstavlja prej omenjenih 8 števcev. Matrika se v osnovni obliki zgradi v dveh korakih:

1. Po formuli (3.3) se določijo števci pripetja vseh celic na korensko vozlišče.

2. Za vsako pripetje celice na vozlišče k, ki ni korensko, se določi sprememba števcev, ki ponazarja spremembo iz $P(D_{kj}|A(T)_{k\sigma_j} = 0)$ v $P(D_{kj}|A(T)_{k\sigma_j} = 1)$.

Opazimo lahko, da zaradi uporabe števcev namesto konkretnih verjetnosti matrika ni odvisna od α in trenutne β, prav tako pa v tej obliki matrika tudi ni odvisna od specifičnega mutacijskega drevesa, temveč le od podanih mutacijskih profilov vzorčenih celic. V tej obliki se tako matrika zgradi enkratno ob začetku izvajanja algoritma, kar vpliva tudi na hitrost natančnega ocenjevanja mutacijskih dreves. Za vsako natančno ocenjevanje se matriko `fast_matrix` pridobi v treh korakih:

1. S časovno relativno hitrim ukazom `np.copy` se skopira osnovna oblika `mask_matrix`.

2. Za vsako pripetje celice na vozlišče, ki ni korensko, se iz pripetja iste celice na starševsko vozlišče in spremembe števcev določi števce, ki ponazarjajo produkt oblike (3.2).

3. Izračuna se skalarni produkt 3-dimenzionalne matrike `mask_matrix` z vektorjem z 8 komponentami (vsaka komponenta predstavlja eno izmed verjetnosti $P(D_{ij} = x|A(T)_{i\sigma_j} = y)$).

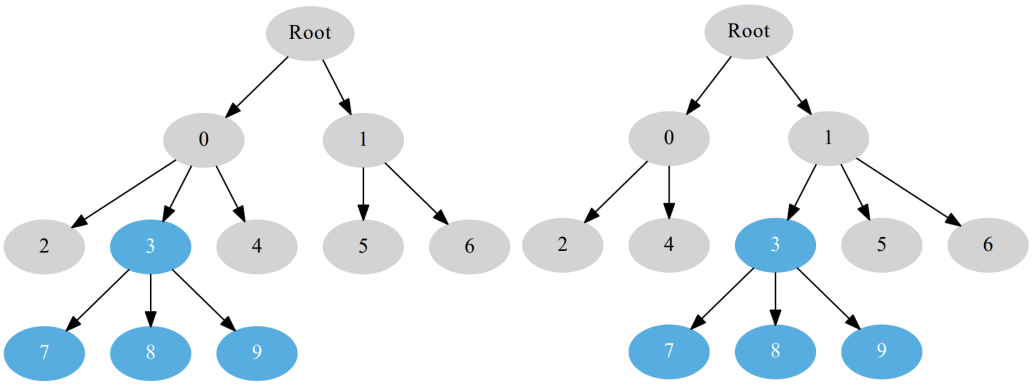
Uporaba `fast_matrix` in `mask_matrix` ima dve pozitivni posledici:

1. Računanje časovno zahtevne funkcije `np.exp` se zgodi samo enkrat namesto $((n + 1) \times m)$-krat.

2. Omogočeno je delno ocenjevanje mutacijskih dreves.

Delno ocenjevanje mutacijskega drevesa pomeni, da se produkti tipa (3.2) ponovno poračunajo samo za tista vozlišča mutacijskega drevesa, na katera je vplivala sprememba drevesa. Da je delno ocenjevanje sploh možno, je potrebno imeti shranjeno matriko `fast_matrix` oziroma `mask_matrix`, ki ustreza mutacijskemu drevesu (v primeru `fast_matrix` tudi β) pred zadnjim premikom. Ocenjevanje mutacijskega drevesa z `mask_matrix` je veliko počasnejše od tistega s `fast_matrix`. Poleg tega se natančno ocenjevanje zgodi relativno redko in sicer takrat, ko je trenutno mutacijsko drevo glede na „nenatančno“ oceno zelo blizu oceni do sedaj najboljšega obiskanega mutacijskega drevesa. Posledično je redko natančno ocenjevanje brez delnega ocenjevanja hitrejše od delnega natančnega ocenjevanja na vsakem koraku. Delno ocenjevanje se tako v praksi uporablja samo pri `fast_matrix`.

Oglejmo si primer delnega ocenjevanja za premik drevesa „prune and reattach“. Primer takega premika lahko vidimo na sliki 3.1. Opozorimo, da tu skladno z našo implementacijo vozlišča označujemo s števili začenši z 0 za razliko od formule (3.1), v kateri avtorji metode vozlišča označujejo začenši z 1. Pri delnem ocenjevanju se izkaže, da je bolj smiselno uporabljati DFT



Slika 3.1: Primer mutacijskega drevesa pred premikom tipa „prune and reattach“ (levo) in po premiku (desno). Z modro barvo so označena vozlišča, katerim so se pri premiku spremenili vnosi v matriki `fast_matrix` oziroma produkti tipa (3.2).

namesto BFT, in sicer tako različico DFT, pri katerem je starševsko vozlišče obiskano pred svojimi otroci (angl. *pre-order*). Vrstni red vozlišč drevesa iz slike 3.1 je po premiku z uporabo takšnega vrstnega reda obiskovanja vozlišč enak:

`Root, 0, 2, 4, 1, 3, 7, 8, 9, 5, 6.`

Ker morajo biti ponovno ovrednotena zgolj vozlišča `3, 7, 8, 9`, lahko opazimo, da se omenjena vozlišča pri DFT nahajajo skupaj ter da je prvo izmed teh vozlišč navedeno tisto vozlišče, ki je bilo med premikom pripeto na novo vozlišče (vozlišče `3`). Ob poznavanju vozlišča, ki bo pripeto na novo vozlišče in velikosti poddrevesa, lahko ovrednotimo vozlišča, na katera je premik vplival, z enim obhodom DFT. Število naslednikov vozlišča i (vključno s sabo) se da preposto dobiti v $O(n)$ časa z vsoto i-te vrstice matrike prednikov.

Delno ocenjevanje mutacijskih dreves smo implementirali za premike „prune and reattach“, „swap node labels“ in „swap subtrees“. Pri slednjih dveh premikih je potrebno ponovno ovrednotiti dve poddrevesi mutacijskega drevesa namesto zgolj enega. Sicer pa je delno ocenjevanje za slednja dva premika zelo podobno opisanemu, zato se tu ne spuščamo v podrobnosti delnega

ocenjevanja mutacijskih dreves za vse premike.

Delno ocenjevanje dreves v primeru uporabe marginalizacije točk pripetja celic na vozlišča deluje hitreje od vsakokratnega ovrednotenja celotnega drevesa, a na koeficient pohitritve močno vpliva dejstvo, da je potrebno tako po ovrednotenju celotnega drevesa kot tudi po delnem ovrednotenju drevesa izračunati oceno z uporabo zunanjega množenja in seštevanja iz formule (3.1). Pri implementaciji iskanja dreves ML, bi se mutacijsko drevo iz matrike `fast_matrix` ocenilo kot

```
np.sum(np.amax(fast_matrix, axis=0)),
```

kar je časovno manj potratno od prej omenjenega ocenjevanja. Pri iskanju dreves ML bi bil torej koeficient pohitritve večji kot pri uporabi marginalizacije. Velja omeniti, da v verziji 0.39.0 programskega paketa Numba (zadnja verzija v času pisanja diplome) ni omogočeno prevajanje funkcije `np.amax` z uporabo dodatnih parametrov v načinu „nopython“, kar je le ena izmed mnogih omejitev, ki jih je potrebno upoštevati ob implementaciji. V našem primeru bi bilo zato potrebno največjo vrednost vsakega stolpca matrike poiskati v zanki.

Vpliv delnega ocenjevanja mutacijskih dreves na hitrost ocenjevanja z uporabo marginalizacije točk pripetja celic na vozlišča opišemo v poglavju 4.3.

3.3 Razširitev programske opreme z izračunom ESS

V poglavju 2.1 opišemo problematiko, ki nastane pri vzorčenju z uporabo metod MCMC. V naši implementaciji se problematike lotimo z izračunom ESS (angl. *effective sample size*), s katerim lahko ovrednotimo stopnjo avtokoreliranosti. ESS oceni število neodvisnih vzorcev, katerim je ekvivalentno naše vzorčenje [7].

Avtorji metode so razmerja prej omenjenih treh premikov mutacijskih dreves ocenjevali s številom obiskanih stanj, po katerih algoritem najde mu-

tacijsko drevo ML. Z računanjem ESS lahko razmerja premikov ocenimo na način, ki je bolj primeren pri ugotavljanju aposteriornih porazdelitev. Poleg tega se lahko prepričamo, da je veriga MCMC zadosti dolga, da imamo zadosti neodvisnih vzorcev.

V programskem jeziku Python smo implementirali računanje ESS za β ter za mutacijska drevesa. Izračun ESS iz vektorja realnih števil smo opravili s programsko opremo Žige Sajovica, ki med drugim omogoča tudi računanje ESS[3]. Za β se lahko ESS izračuna kar neposredno iz njenih vzorcev, saj so vzorci realna števila. Pri mutacijskih drevesih pa postopek ni tako trivialen.

Tu uporabimo metodo pseudo-ESS [7], pri kateri se iz aposteriorne porazdelitve izbere neko naključno drevo (Lanfear et al. ga imenujejo „focal tree“) in se glede na neko definicijo razdalje izračuna razdaljo izbranega drevesa do vseh dreves v aposteriorni porazdelitvi. Na ta način dobimo vektor realnih števil, iz katerega lahko izračunamo ESS na enak način kot za β. Celoten postopek se ponovi večkrat, kot dokončni ESS pa se vrne mediano vseh izračunanih ESS.

Lanfear et al. za računanje razdalje uporabijo razdalji Path Difference in Robinson–Foulds, ki sta zasnovani za primerjavo filogenetskih dreves. V našem primeru omenjeni razdalji nista primerni, saj nimamo opravka s filogenetskimi drevesi, pač pa z mutacijskimi drevesi. Za računanje ESS tako uporabljamo dve drugačni razdalji, in sicer preprosto razdaljo, ki se uporablja že pri SCITE ter razdaljo SuMoTED [8].

Primeri izračunanih ESS so v poglavju 4.5, kjer so izračunani za β in za mutacijska drevesa z uporabo obeh razdalj.

3.3.1 Preprosta razdalja

Razdalja, ki jo tu imenujemo preprosta, nam glede na dve mutacijski drevesi pove, koliko vozlišč je takih, ki nima istega starša v obeh mutacijskih drevesih. Razdaljo avtorji metode uporabljajo za računanje razdalje trenutnega mutacijskega drevesa v algoritmu do dejanskega mutacijskega drevesa, ki je

[3]https://github.com/ZigaSajovic/mcmcse

generiralo podatke, če le-to mutacijsko drevo poznamo. Razdaljo med dvemi mutacijskimi drevesi se da zaradi enostavnosti tudi relativno hitro izračunati v $O(n)$ časa, če z n označimo število mutacij.

3.3.2 SuMoTED

Razdalja SuMoTED [8] (**Su**btree **Mo**ving **T**ree **E**dit **D**istance) je razdalja za drevesa s korenskim vozliščem, ki imajo vsa vozlišča označena z različnimi oznakami in pri katerih vrstni red otrok vozlišča ni pomemben. Če z n označimo število vozlišč, ima izračun razdalje časovno zahtevnost $O(n^2)$. Razdalja ponazarja minimalno število lokalnih premikov, ki so potrebni, da se prvo drevo preoblikuje v drugega. Lokalen premik označuje premik tipa „prune and reattach“, pri katerem velja omejitev, da se mora poddrevo pripeti ali na eno izmed svojih trenutnih sestrskih vozlišč (vozlišča z enakim staršem) ali pa na starša svojega trenutnega starša. Premiki so uteženi glede na število vozlišč v poddrevesu, ki je premaknjeno.

Za potrebe izračuna ESS smo izračun SuMoTED pohitrili z učinkovitejšo implementacijo v programskem jeziku Python, kot tudi z uporabo programskih paketov NumPy in Numba. Vpliv učinkovitejše implementacije na hitrost delovanja je opisan v poglavju 4.6.

3.4 Nov premik drevesa

Poleg obstoječih treh premikov drevesa smo implementirali nov premik drevesa, ki smo ga poimenovali „remove and insert“. Premik iz drevesa odstrani eno samo vozlišče in ga nato vstavi na neko drugo mesto v drevesu. Nov premik predlagamo na osnovi ugotovitve, da noben izmed obstoječih premikov ne premika samo enega vozlišča na poljubno mesto. Premik „swap node labels“ sicer naredi podoben premik, a se pri tem premiku dejansko zgodita dva premika, kar pa lahko privede do tega, da je samo eden izmed dveh dejansko ugoden. Premika „prune and reattach“ in „swap subtrees“ sicer lahko premikata samo po eno vozlišče oziroma zamenjata dve posamezni vozlišči,

a se premik vedno zgodi pri listih drevesa.

Na sliki 3.2 lahko vidimo primer premika „remove and insert“. Če število vozlišč brez korenskega vozlišča označimo z n, lahko premik opišemo z naslednjimi koraki:

1. Enakomerno slučajno izberemo vozlišče $i \in \{0, 1, \ldots, n-1\}$, ki ga odstranimo iz drevesa. Z p označimo starša vozlišča i. Na sliki 3.2 je vozlišče i označeno z rdečo barvo, vozlišče p pa z modro barvo.

2. Vozlišča množice $W = otroci(i)$ postanejo otroci vozlišča p. Z množico X označimo vse otroke vozlišča p po tem koraku.

3. Enakomerno slučajno izberemo vozlišče $j \in \{0, 1, \ldots, n-1\} \backslash \{i\} \cup \{Root\}$. Na sliki 3.2 je to vozlišče označeno z vijolično barvo.

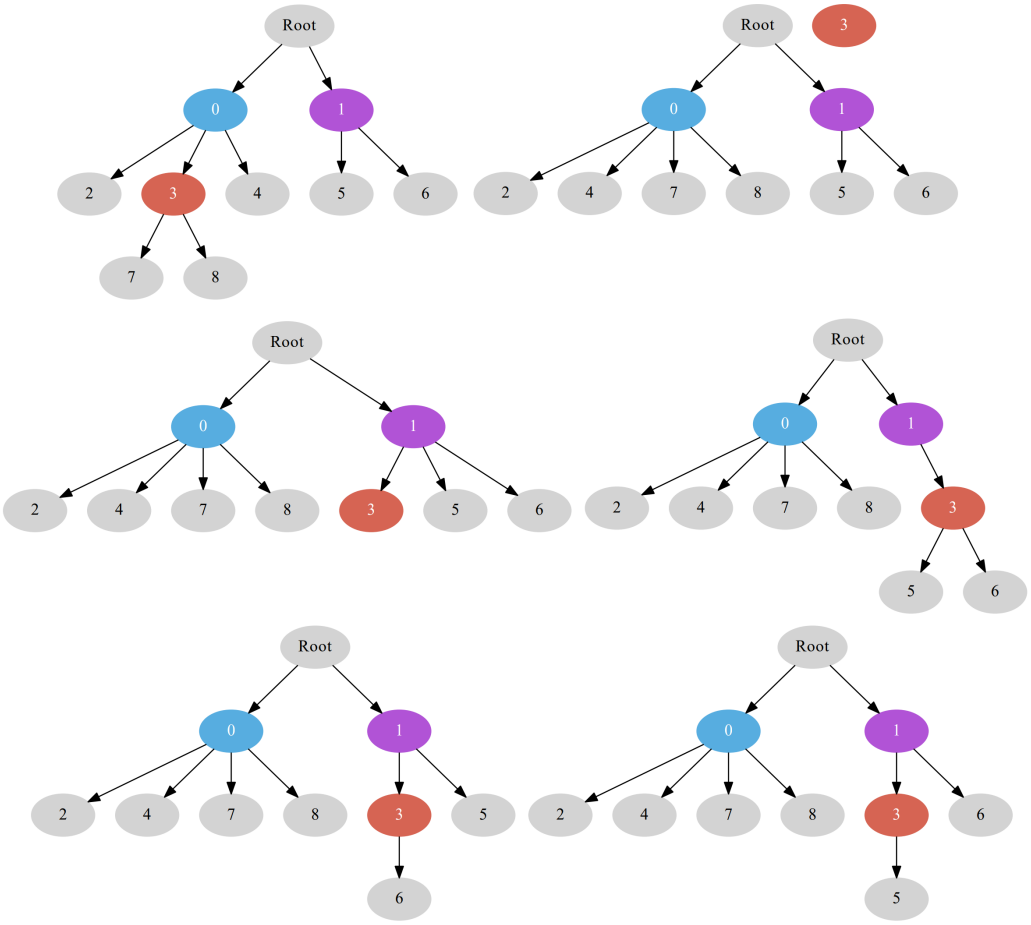
4. Enakomerno slučajno izberemo podmnožico $Z \subseteq Y$, $Y = otroci(j)$. Vozlišča iz podmnožice pripnemo na vozlišče i, tako da niso več pripeta na vozlišče j.

5. Na vozlišče j pripnemo vozlišče i.

Tako kot prej, tudi tu skladno z našo implementacijo vozlišča označujemo s števili začenši z 0.

Lahko se zgodi, da velja $j = p$, v nekaterih primerih pa lahko poleg omenjene enakosti velja tudi $W = Z$. V slednjem primeru je drevo po premiku enako drevesu pred premikom. Ta lastnost zagotavlja aperiodičnost markovske verige ob uporabi tega premika. Pri uporabi novega premika je zagotovljena tudi nerazcepnost markovske verige, saj premik zagotavlja, da lahko v končnem številu korakov iz kateregakoli mutacijskega drevesa dosežemo katerokoli drugo mutacijsko drevo. Pri premiku verjetnost prehoda iz prvega drevesa v drugo drevo ni nujno enaka verjetnosti prehoda iz drugega v prvo, zato je potreben izračun korekcijskega koeficienta, ki ga opišemo v formuli (2.3). Če prvo drevo označimo s T_1 in drugo s T_2, lahko verjetnost premika iz prvega drevesa v drugega zapišemo kot

$$q(T_2|T_1) = \frac{1}{n} \times \frac{1}{n} \times \frac{1}{2^{|Y|}}. \tag{3.4}$$



Slika 3.2: Zgoraj levo: drevo pred premikom. Zgoraj desno: drevo po prvih dveh korakih premika. Spodnje štiri slike: štiri možni načini vstavitve izbranega vozlišča nazaj v drevo.

Prvi ulomek ponazarja verjetnost izbire vozlišča i, drugi verjetnost izbire vozlišča j, tretji pa verjetnost izbire določene podmnožice vozlišč Y. Verjetnost premika iz drugega drevesa v prvo drevo lahko podobno zapišemo kot

$$q(T_1|T_2) = \frac{1}{n} \times \frac{1}{n} \times \frac{1}{2^{|X|}}. \tag{3.5}$$

Korekcijski koeficient lahko nato izračunamo kot

$$c = \frac{q(T_1|T_2)}{q(T_2|T_1)} = 2^{|Y|-|X|}. \tag{3.6}$$

Ker je zagotovljena simetričnost verjetnosti prehoda in premik sam po sebi zagotavlja aperiodičnost ter nerazcepnost markovske verige, lahko z uporabo zgolj tega premika ali v kombinaciji z obstoječimi premiki z obiskanimi stanji markovske verige modeliramo aposteriorne porazdelitve mutacijskih dreves in β [1].

Vplive uporabe novega premika opišemo v poglavju 4.5.

Poglavje 4

Poskusi in rezultati

V poglavju s terko (a, b, c, d) označimo razmerje premikov mutacijskih dreves, pri katerem se a časa izbere premik „prune and reattach“, b časa „swap node labels“, c časa „swap subtrees“ in d časa „remove and insert“.

4.1 Podatki

V tabeli 4.1 so povzete ključne lastnosti uporabljenih podatkovnih množic. Za podatkovno množico Xu avtorji metode navajajo, da razlikuje med homozigotnimi in heterozigotnimi mutacijami. Podatkovna množica, ki je dosegljiva na repozitoriju implementacije ne vsebuje številke 2, kar pomeni, da homozigotna mutacija nikoli ni bila zaznana. Poleg tega avtorji metode ne navajajo podatka o verjetnosti $P(D_{ij} = 2|E_{ij} = 0)$, ki je eden izmed parametrov metode (v njihovi implementaciji označen s `-cc`). Posledično pri omenjeni podatkovni množici homozigotnih mutacij nismo upoštevali.

Podatkovno množico MissionBio smo pridobili s pomočjo programske opreme Tapestri Insights in njej priloženih podatkovnih množic[1]. Uporabili smo edino podatkovno množico, ki je imela specificirano verjetnost β. Ker verjetnost α ni bila navedena, smo uporabili relativno majhno verjetnost 1×10^{-6}.

[1] https://missionbio.com/panels/software/

Pod. množica	n	m	β	α	hom. mut.
Hou18 [3, 4]	18	58	0.4309	6.04×10^{-5}	da
Xu [14, 4]	35	17	0.1643	2.67×10^{-5}	ne
Navin [13, 4]	40	47	0.0972	1.24×10^{-6}	ne
MissionBio	20	8954	0.0697	1×10^{-6}	ne

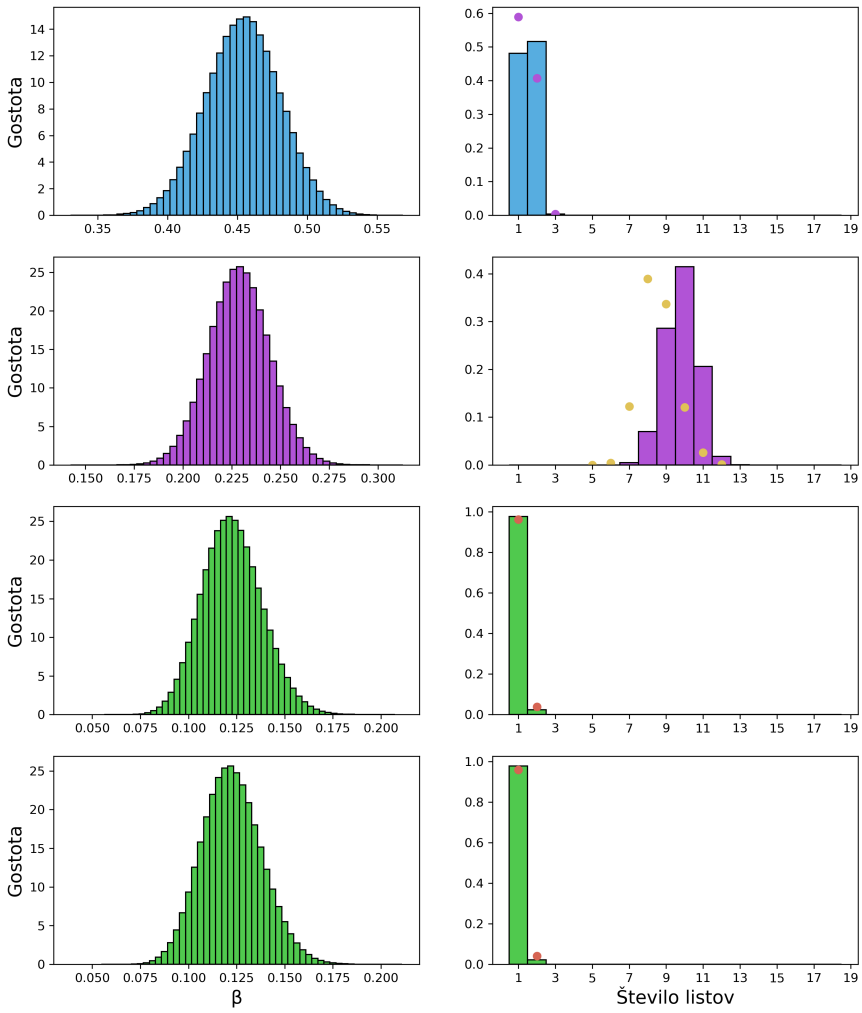
Tabela 4.1: Povzetek uporabljenih podatkovnih množic. Z n je označeno število mutacij, z m število vzorčenih celic, z α ocenjena verjetnost napačno zaznane mutacije, z β ocenjena verjetnost nezaznane mutacije, s hom. mut. pa, ali podatkovna množica ločuje med homozigotnimi in heterozigotnimi mutacijami.

4.2 Pravilnost implementacije

Pravilnost delovanja smo preverili s primerjavo dobljenih rezultatov izvajanja algoritma za različne podatkovne množice z rezultati avtorjev metode SCITE. Primerjali smo aposteriorne porazdelitve razvejanosti dreves in β ter mutacijska drevesa MAP in β MAP.

Za izvedbo primerjave je bilo potrebno sklepati na nekatere parametre algoritma, saj avtorji metode niso navedli točnih parametrov za vse podatkovne množice. Premik β smo skladno s podatkovno množico Hou18 tudi pri ostalih dveh podatkovnih množicah nastavili tako, da se zgodi 10 % časa, medtem ko se 90 % časa zgodi premik oziroma preoblikovanje mutacijskega drevesa. Podatek o verjetnosti $P(D_{ij} = 2|E_{ij} = 0)$ za podatkovno množico Hou18 smo pridobili z repozitorija implementacije avtorjev. Količino, ki jo v enačbi (2.1) označimo z x_β in je eden izmed parametrov metode, smo skladno z implementacijo avtorjev nastavili na 10, kjer vrednost ni bila navedena.

Iz slike 4.1 in tabele 4.2 lahko ugotovimo, da so aposteriorne porazdelitve za podatkovni množici Hou18 in Navin skladne s tistimi, ki jih navajajo avtorji metode, česar pa ne moremo trditi za podatkovno množico Xu. Odločili smo se, da z enakimi parametri poženemo program avtorjev metode in tako



Slika 4.1: Aposteriorna porazdelitev β (levo) in števila listov (desno) za podatkovne množico Hou18 (1. vrstica), Navin (2. vrstica) in Xu (3. in 4. vrstica). Porazdelitve ne vsebujejo prve četrtine vzorcev, saj predstavljajo fazo „burn-in“. V desnih histogramih stolpci predstavljajo porazdelitev števila listov, ko je β znana, medtem ko krogci predstavljajo isto porazdelitev, ko algoritem ugotavlja tudi vrednost β.

	Hou18	Navin	Xu (Python)	Xu (C++)
μ	0.454	0.229	0.123	0.123
σ	0.027	0.016	0.016	0.016

Tabela 4.2: Aritmetične sredine in standardni odkloni aposteriornih porazdelitev β iz slike 4.1.

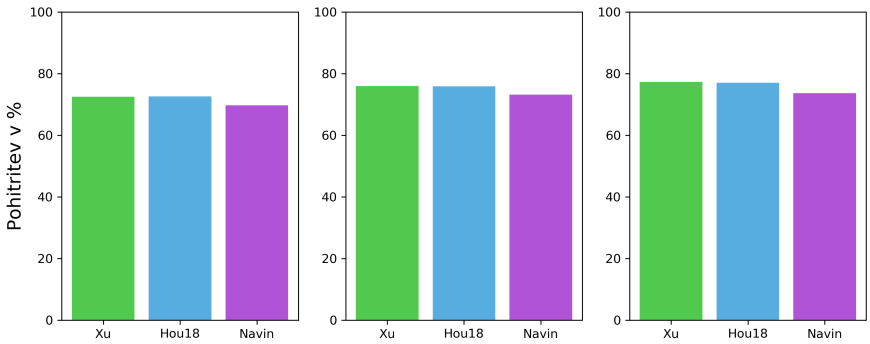
preverimo skladnost delovanja. Naša implementacija (3. vrstica na sliki 4.1) se je tako tudi na podatkovni množici Xu izkazala za skladno implementaciji avtorjev metode (4. vrstica na sliki 4.1). Odstopanje je verjetno posledica uporabe drugačnih parametrov, ki pa so, kot že omenjeno, ponekod zgolj delno opisani.

Pri podatkovni množici Hou18 smo z našo implementacijo prišli do enakih MAP dreves tako, ko je β fiksna, kot tudi, ko jo algoritem ugotavlja poleg mutacijskega drevesa. Dobili smo tudi enako MAP verjetnost β.

Pri podatkovni množici Navin smo v primeru fiksne β dobili več enako verjetnih dreves, ki imajo logaritemsko oceno -571.7, ki je enaka logaritemski oceni drevesa, ki so jo dobili avtorji metode. Ko algoritem poleg mutacijskega drevesa ugotavlja tudi β, se drevo le malo razlikuje od tistega, ki so ga dobili avtorji metode. Z MAP verjetnostjo $\beta = 0.226$ logaritemska skupna ocena drevesa in β MAP znaša -521.5, kar je enako oceni, ki so jo dobili avtorji metode.

Mutacijska drevesa MAP in β MAP pri podatkovni množici Xu zaradi prej omenjene problematike nepoznavanja parametrov niso uporabna za ocenjevanje korektnosti implementacije.

Sodeč po rezultatih testiranja sklepamo, da naša implementacija metode SCITE v programskem jeziku Python deluje skladno z implementacijo avtorjev v programskem jeziku C++.

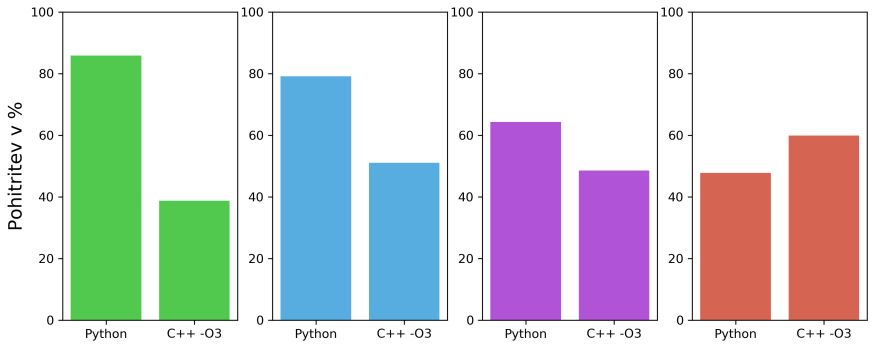


Slika 4.2: V % izražena pohitritev pri uporabi delnega ocenjevanja mutacijskih dreves v primerjavi z navadnim ocenjevanjem za premik „prune and reattach“ (levo), „swap node labels“ (sredina) in „swap subtrees“ (desno).

4.3 Delno ocenjevanje mutacijskih dreves

Vpliv delnega ocenjevanja mutacijskih dreves smo preverili z izvajanjem algoritma SCITE na podatkovnih množicah Xu, Hou18 in Navin. Med izvajanjem smo pri vsakem ocenjevanju drevesa drevo ocenili tako z navadnim pristopom, pri katerem je ocenjeno celotno drevo, kot tudi z delnim ocenjevanjem. Uporabili smo razmerje premikov (0.55, 0.4, 0.05, 0), ki ga predlagajo avtorji metode. Na sliki 4.2 lahko opazimo, da do največje pohitritve pride pri premiku „prune and reattach“, saj se ponovno ovrednoti samo eno poddrevo, medtem ko se pri ostalih dveh premikih ponovno ovrednotita dve poddrevesi. Opazimo lahko tudi, da je vpliv pohitritve najbolj opazen pri podatkovni množici Navin, kar bi lahko pripisali v povprečju večji razvejanosti dreves (slika 4.1).

Večja razvejanost dreves namreč pomeni v povprečju manjše število vozlišč v naključnem poddrevesu. Pri podatkovnih množicah, ki bi nakazovale na bolj razvejana mutacijska drevsa, bi imela uporaba delnega ocenjevanja mutacijskih dreves tako lahko znatnejši vpliv.



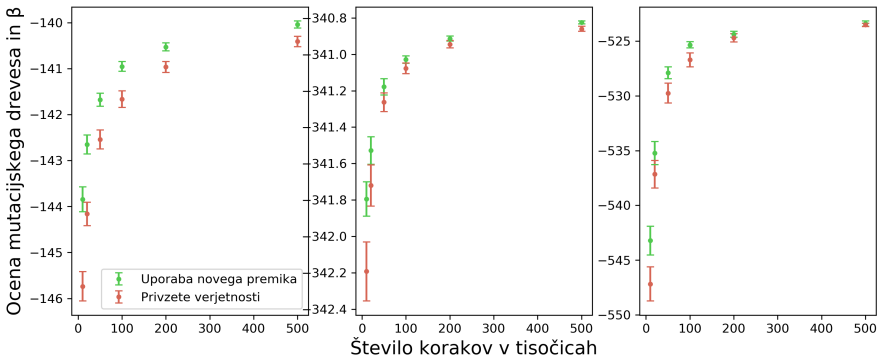
Slika 4.3: Primerjava hitrosti izvajanja za različne podatkovne množice, od leve proti desni: Xu, Hou18, Navin, MissionBio. Pohitritev 100 % predstavlja čas izvajanja programske opreme avtorjev metode s prevajanjem na način `g++ *.cpp -o scite`. Oznaka `C++ -O3` pomeni spremembo omenjenega načina prevajanja z uporabo optimizacijske zastavice `-O3`.

4.4 Hitrost implementacije

Na sliki 4.3 so podatkovne množice od leve proti desni urejene naraščajoče glede na velikost podatkovne množice (z velikostjo podatkovne množice pojmujemo količino $n \times m$ iz tabele 4.1). V vseh primerih je naša implementacija hitrejša od implementacije avtorjev ob uporabi navadnega prevajanja programske opreme. Opazimo lahko, da z večjimi podatkovnimi množicami naša implementacija deluje hitreje v primerjavi z implementacijo avtorjev metode ter da vpliv optimizacijske zastavice `-O3` pada. Naša implementacija torej deluje primerljivo hitro v primerjavi z implementacijo avtorjev metode oziroma boljše v primeru večjih podatkovnih množic.

4.5 Uporaba novega premika

Predvidevali smo, da bo premik „remove and insert“ najbolje zamenjal premik „swap node labels“, v praksi pa se izkaže, da se splača zamenjati premik



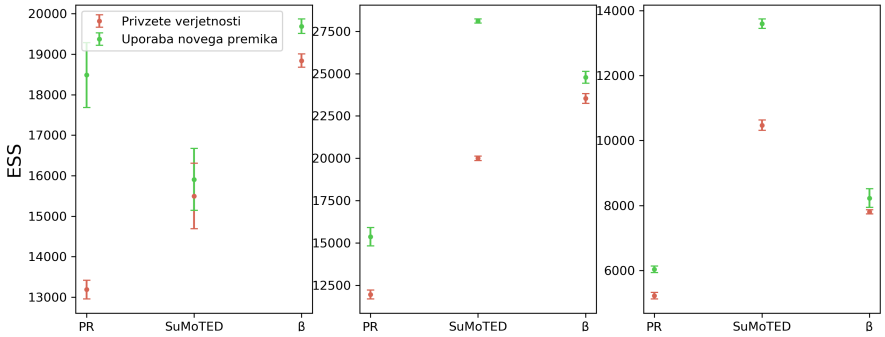
Slika 4.4: Aritmetična sredina ocene do tedaj najboljše komibnacije mutacijskega drevesa in β ter standardna napaka aritmetične sredine za podatkovno množico Xu (levo), Hou18 (sredina) in Navin (desno).

„prune and reattach“. Ugotovitev nakazuje, da je premik, ki ne spreminja strukture drevesa pomemben ter da premik „swap subtrees“ lahko nadomesti premik „prune and reattach“.

Novo razmerje premikov (0, 0.4, 0.05, 0.55) smo primerjali s prej omenjenim razmerjem (0.55, 0.4, 0.05, 0). Poudariti velja, da smo temeljito preiskali samo vplive uporabe tega razmerja premikov drevesa, vendar pa lahko obstaja tudi bolj učinkovito razmerje.

Razmerja smo najprej primerjali na podlagi povprečne ocene do tedaj najboljše kombinacije mutacijskega drevesa in β po 10000, 20000, 50000, 100000, 200000 in 500000 korakih. Za vsako podatkovno množico smo poskus ponovili 50-krat. Pri poganjanju algoritma smo uporabili enake parametre kot pri testiranju pravilnosti delovanja, ki je opisano v poglavju 4.2, razen razmerij premikov. Na sliki 4.4 lahko vidimo, da novo razmerje premikov pri vseh treh podatkovnih množicah algoritem vodi v hitrejše odkritje boljših kombinacij dreves in β.

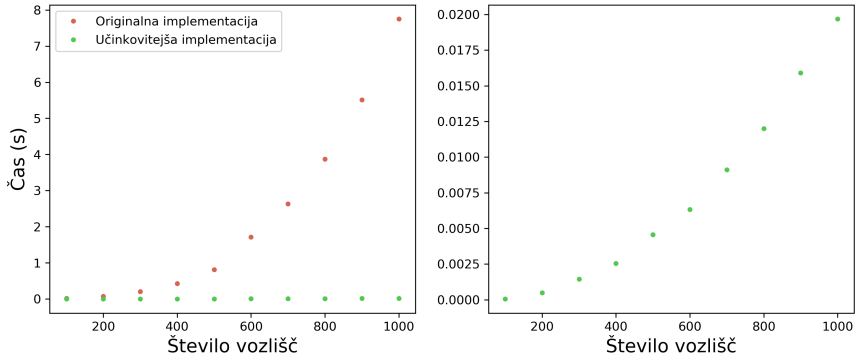
Razmerja smo primerjali tudi na podlagi ESS aposteriornih porazdelitev mutacijskih dreves in β. Za mutacijska drevesa je bil ESS izračunan tako z uporabo preproste razdalje kot tudi z uproabo SuMoTED. Aposteriorne



Slika 4.5: Aritmetična sredina izračunanih ESS ter standardna napaka aritmetične sredine za podatkovno množico Xu (levo), Hou18 (sredina) in Navin (desno). Kratica PR označuje uporabo preproste razdalje pri računanju ESS za mutacijska drevesa.

porazdelitve so pridobljene s poganjanjem algoritma 10^7 korakov z enakimi parametri kot prej. Prvih 25 % vzorcev predstavlja fazo „burn-in“, zato omenjene vzorce odstranimo iz aposteriorne porazdelitve. Ker izračun ESS traja relativno veliko časa, smo vsak poskus ponovili le 4-krat. Pri vseh podatkovnih množicah je bilo pri izračunu ESS za mutacijska drevesa izbranih 100 dreves (angl. *focal tree*) pri uporabi preproste razdalje in 30 dreves pri uporabi SuMoTED, razen pri podatkovni množici Hou18, pri kateri je bilo tudi v tem primeru izbranih 100 dreves. Na sliki 4.5 lahko vidimo, da premik drevesa „remove and insert“ pozitivno vpliva tudi na ESS, saj je pri vseh podatkovnih množicah višji ali pa vsaj primerljiv z ESS, ki ga dobimo z uporabo privzetih razmerij premikov drevesa.

Glede na dobljene rezultate povzemamo, da ima uporaba novega premika drevesa pozitivne vplive na delovanje algoritma, ob tem pa še enkrat poudarjamo, da bi bilo potrebno podrobneje raziskati ostale kombinacije premikov drevesa, ki bi morda delovale še boljše.



Slika 4.6: Aritmetična sredina časov, potrebnih za izračun razdalje za obe implementaciji (levo) ter samo za učinkovitejšo implementacijo (desno).

4.6 Učinkovitejša implementacija SuMoTED

Iz slike 4.6 lahko vidimo, da smo uspešno pohitrili izračun razdalje SuMoTED. Pri primerjavi časov smo pri obeh implementacijah merili čas izvajanja funkcije `distance`, pri čemer funkcija originalne implementacije kot parametre sprejme drevesa zapisana v obliki matrik sosednosti (angl. *adjacency matrix*), pohitrena implementacija pa kot parametre sprejme drevesa zapisana v obliki matrik prednikov.

Poglavje 5

Zaključek

V diplomski nalogi smo implementirali del funkcionalnosti metode SCITE [4]. Kljub uporabi načeloma počasnejšega programskega jezika naša metoda deluje primerljivo hitro v primerjavi z implementacijo avtorjev metode. Predlagali smo delno ocenjevanje mutacijskih dreves, s katerim smo dosegli boljšo primerljivost v hitrosti izvajanja. Omenjena izboljšava bi lahko pripomogla tudi k še hitrejšemu izvajanju v programskem jeziku C++, saj ni specifična za programski jezik Python. Nov premik drevesa „remove and insert“ glede na rezultate testiranja pripomore k hitrejšemu odkritju boljših kombinacij mutacijskih dreves MAP in β MAP ter pozitivno vpliva na ESS aposteriornih porazdelitev. Dodana možnost izračuna ESS omogoča boljše ovrednotenje dobljenih rezultatov izvajanja algoritma.

V prihodnosti se lahko z natančno analizo uporabe novega premika ugotovi novo razmerje premikov s katerim se boljše kombinacije mutacijskih dreves MAP in β MAP dosežejo še hitreje oziroma je ESS aposteriornih porazdelitev še višji. Lanfear et. al [7] poleg uporabljenega pristopa pseudo-ESS za računanje ESS filogenetskih dreves predlagajo tudi pristop approximate-ESS. Z implementacijo slednjega pristopa bi lahko ESS bolje ovrednotili, saj bi imeli za njegov izračun na voljo dva pristopa. Pri uporabi programskega paketa Numba za prevajanje programske kode ne izkoriščamo možnosti paralelizacije operacij z uporabo parametra `parallel=True` v dekoratorju

funkcije. Z ustreznim preoblikovanjem programske kode, ki se prevaja z uporabo programskega paketa Numba, bi lahko s paralelizacijo operacij dosegli občutno pohitritev. Naša programska oprema trenutno ne omogoča iskanja mutacijskih dreves ML, a implementacija razširitve ne bi bila zahtevna. Implementacijo bi lahko v prihodnosti nadgradili tudi z implementacijo delnega ocenjevanja mutacijskih dreves za premik „remove and insert“ ter dodatnih funkcionalnosti, ki jih prinaša razširjena metoda ∞SCITE.

Literatura

[1] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[2] Cameron Davidson-Pilon. *Bayesian methods for hackers: probabilistic programming and Bayesian inference*. Addison-Wesley Professional, 2015.

[3] Yong Hou, Luting Song, Ping Zhu, Bo Zhang, Ye Tao, Xun Xu, Fuqiang Li, Kui Wu, Jie Liang, Di Shao, et al. Single-cell exome sequencing and monoclonal evolution of a JAK2-negative myeloproliferative neoplasm. *Cell*, 148(5):873–885, 2012.

[4] Katharina Jahn, Jack Kuipers, and Niko Beerenwinkel. Tree inference for single-cell data. *Genome Biology*, 17(1):86, 2016.

[5] Kyung In Kim and Richard Simon. Using single cell sequencing data to model the evolutionary history of a tumor. *BMC Bioinformatics*, 15(1):27, 2014.

[6] Jack Kuipers, Katharina Jahn, Benjamin J Raphael, and Niko Beerenwinkel. Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors. *Genome Research*, 2017.

[7] Robert Lanfear, Xia Hua, and Dan L Warren. Estimating the effective sample size of tree topologies from Bayesian phylogenetic analyses. *Genome Biology and Evolution*, 8(8):2319–2332, 2016.

[8] Matt McVicar, Benjamin Sach, Cédric Mesnage, Jefrey Lijffijt, Eirini Spyropoulou, and Tijl De Bie. SuMoTED: An intuitive edit distance between rooted unordered uniquely-labelled trees. *Pattern Recognition Letters*, 79:52–59, 2016.

[9] Edith M Ross and Florian Markowetz. OncoNEM: inferring tumor evolution from single-cell sequencing data. *Genome Biology*, 17(1):69, 2016.

[10] Sohrab Salehi, Adi Steif, Andrew Roth, Samuel Aparicio, Alexandre Bouchard-Côté, and Sohrab P Shah. ddClone: joint statistical inference of clonal populations from single cell and bulk tumour sequencing data. *Genome Biology*, 18(1):44, 2017.

[11] Michael R Stratton, Peter J Campbell, and P Andrew Futreal. The cancer genome. *Nature*, 458(7239):719, 2009.

[12] Charles Swanton. Intratumor heterogeneity: evolution through space and time. *Cancer Research*, 2012.

[13] Yong Wang, Jill Waters, Marco L Leung, Anna Unruh, Whijae Roh, Xiuqing Shi, Ken Chen, Paul Scheet, Selina Vattathil, Han Liang, et al. Clonal evolution in breast cancer revealed by single nucleus genome sequencing. *Nature*, 512(7513):155, 2014.

[14] Xun Xu, Yong Hou, Xuyang Yin, Li Bao, Aifa Tang, Luting Song, Fuqiang Li, Shirley Tsang, Kui Wu, Hanjie Wu, et al. Single-cell exome sequencing reveals single-nucleotide mutation characteristics of a kidney tumor. *Cell*, 148(5):886–895, 2012.

[15] Ke Yuan, Thomas Sakoparnig, Florian Markowetz, and Niko Beerenwinkel. BitPhylogeny: a probabilistic framework for reconstructing intratumor phylogenies. *Genome Biology*, 16(1):36, 2015.

[16] Hamim Zafar, Anthony Tzen, Nicholas Navin, Ken Chen, and Luay Nakhleh. SiFit: inferring tumor trees from single-cell sequencing data under finite-sites models. *Genome Biology*, 18(1):178, 2017.